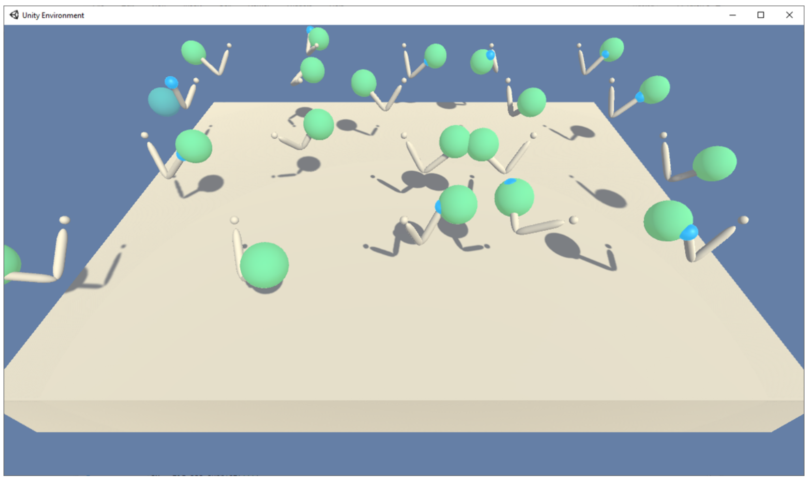
# Udacity Reinformcement Learning Nanodegree: P2 Continuous Control

# Report

In this project, we are working with Reacher environment and set up an agent that learns to control a two-armed robot to follow a target in the environemnt. This repository has the model and the code for training the agent as well as data for a trained agent.



## Introduction

In the environment, a double-jointed arm should move to target locations, and a reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of your agent is to maintain its position at the target location for as many time steps as possible.

In this project, there are two options to choose from. The environment can contain one single robot, or a collection of 20 robots. In the first case, the agent will control the single robot, while in the second case, all robots are controlled. This repository presents a solution for option 2, with 20 agents.

We have an episodic task. After each episode, the rewards that each agent has received are added up, which yields 20 scores. Average of these values will be the score for each episode. The environment is considered solved, when the average over 100 episodes of those average scores is at least 30. During training the data from all agents are collected and used for training.

The observation space consists of 33 real-valued variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four real numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be between -1 and 1. Thus, we have an episodic, continuous control problem to solve.

## Solution method

Deep Deterministic Policy Gradient (DDPG) is used to solve this environment, which is a continuous control problem. In this method, two neural networks are used, one as actor and one as critic. The actor estimates the actor network produces the believed best action for a given state description, which is basically the deterministic policy. In other words, it learns $\mu(s; \theta_\mu) = argmax_a Q(s, a)$. The critic, on the other hand, evaluates the action value function, for the best believed action, i.e. $Q(s, \mu(s, \theta_\mu); \theta_Q)$. Here, $s$ and $a$ denote environment state and actions and $\theta$ the approximation parameters, or the network weights.

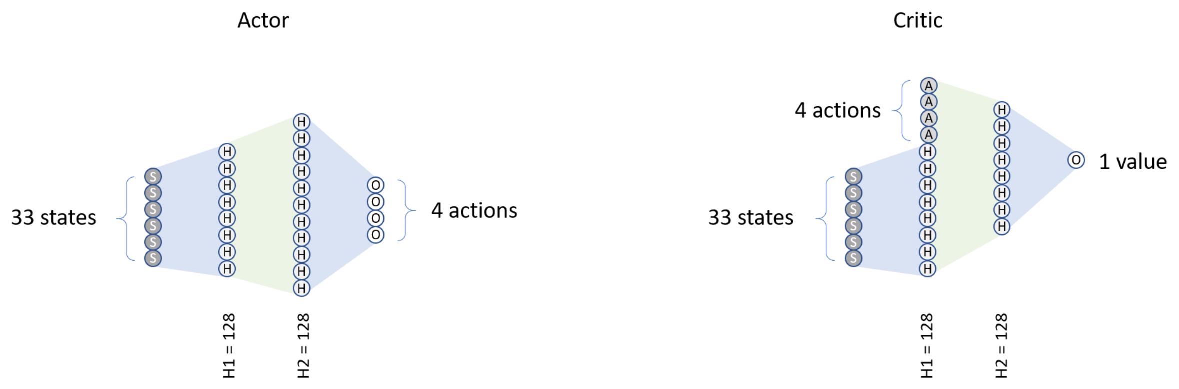
During training, the networks are updated in the following way:

1. Given the current state ($s$), the actor network suggests actions ($a = \mu(s)$).
2. The actions are sent to the environment and new states ($s_{next}$) and rewards ($r$) are recorded.
3. New actions are calculaed for next state by the actor, i.e. $a_{next} = \mu(s_{next})$.
4. A new value for the current state is calculated by adding the obtained reward and discounted estimated reward for the next state, i.e. $Q = r + \gamma \times Q(s_{next}, \mu(s_{next}))$, where $\gamma$ is the discount factor.
5. The critic network is updated to minimize the difference between the difference between the current estimate, $Q(s, a)$ and the above calculated discounted reward.
6. The actor network will be updated so that the reward, predicted by the critic network, will be maximized. That is done by first using the actor network to get the suggested actions and then the critic to estimate the value from the states and believed best action. That means, we maximize $Q(s, \mu(s))$

Similar to DQN, DDPG, is using two networks, local and target. During the learning process for the critic network, the target actor is used to evaluate the actions for the next state and the weights of the local critic network are updated. Similarly, the weights of the local actor network are updated using the local critic network with upated weights. A scheme called soft update to update the target network, i.e. $\theta_{target} = \tau \times \theta_{local} + (1 - \tau) \times \theta_{target}$, where $\tau$ is a relaxation parameter that hould be set to a small value.

My starting point was the implementation of DDPG method in the Pendulum example. The actor network has the state vector as input and action vector as output. Critic network has both state vector and action vector as inputs and estimates the reward. Networks with two hidden layes of 128 were used as shown below:

```
Actor network: Actor(
  (fc1): Linear(in_features=33, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=4, bias=True)
)
Critic network: Critic(
  (fcs1): Linear(in_features=33, out_features=128, bias=True)
  (fc2): Linear(in_features=132, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=1, bias=True)
)
```

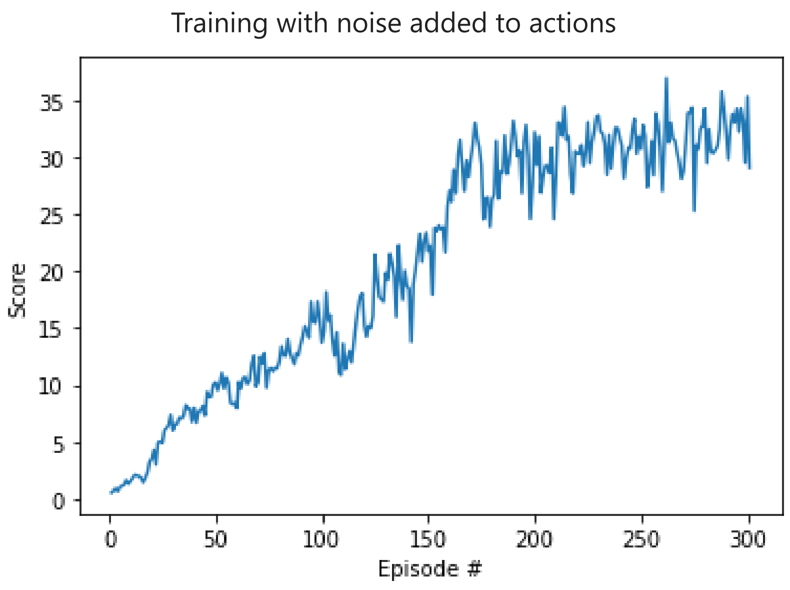


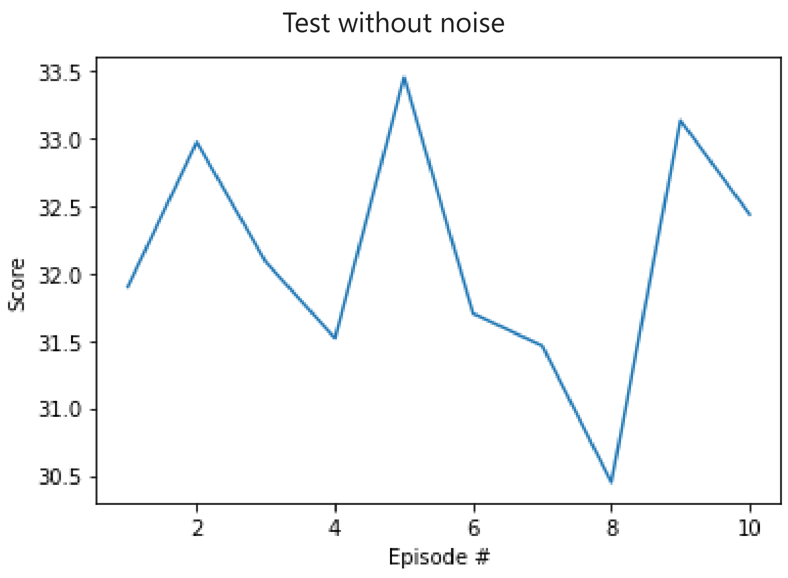
Otherwise the hyper parameters were kept as in ddpg_agent.py:

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 128        # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR_ACTOR = 1e-4         # learning rate of the actor
LR_CRITIC = 1e-3        # learning rate of the critic
WEIGHT_DECAY = 0        # L2 weight decay

`
```

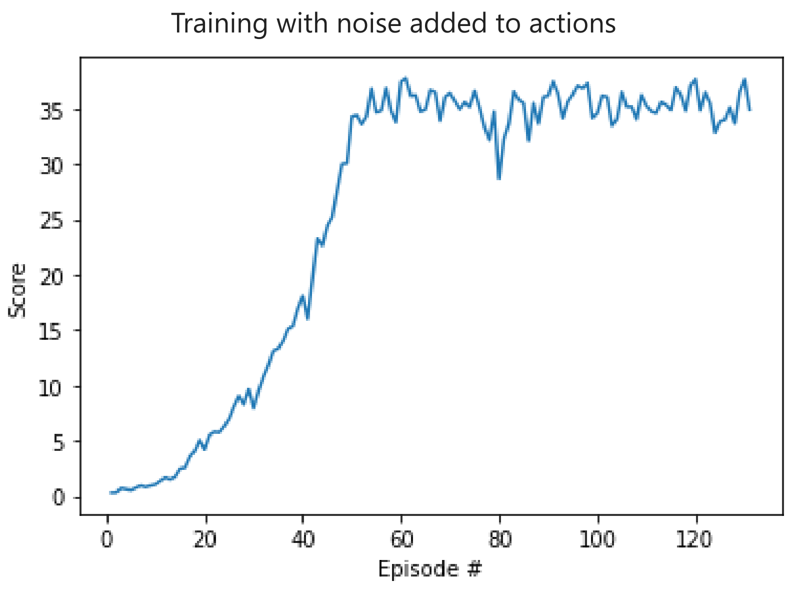
The results from training and testing is hown below:

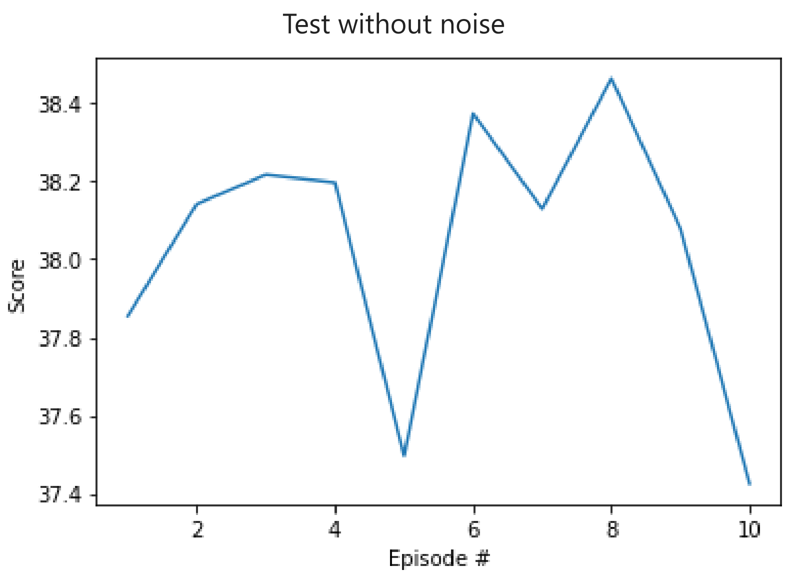




300 episodes needed to solve the environment.

As can be found in the above figures, the target can be achieved after approximately 300 episodes. My other attempts suggested that the training process with these parameters were not very stable. Reduction of learning rate for the critic to the same value as for the actor, i.e. $10^{-4}$ resulted in a more stable process and reaching the target and higher test scores with fewer episodes. Compare the figures above with those below. This value was used in all cases reported below.





130 episodes needed to solve the environment.

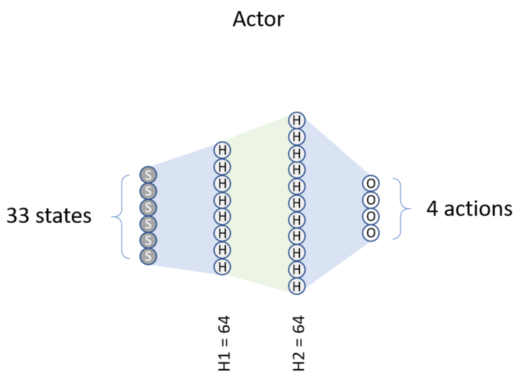
In the results presented below, the learning rate wa set to $10^{-4}$ for both actor and critic networks.

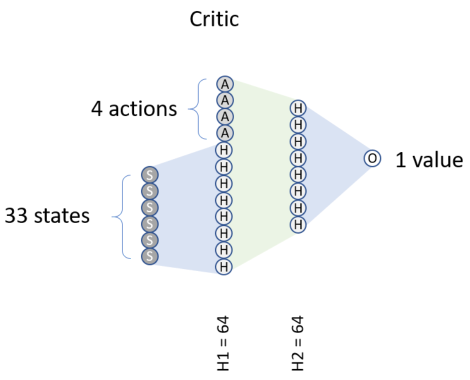
## Training and testing results

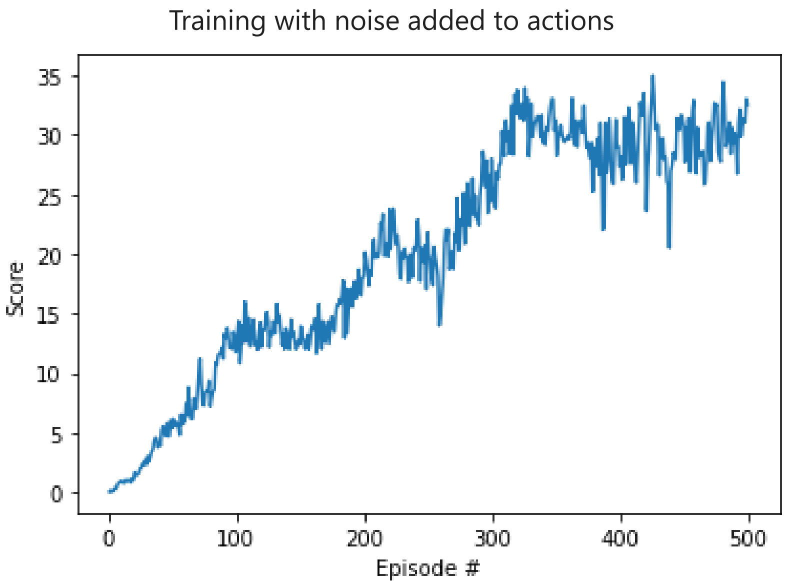
To train the agent, you can run the test using the notebook training_and_report.ipynb. Below figures presented the result after that the average of score in the last 100 episodes has passed 13.

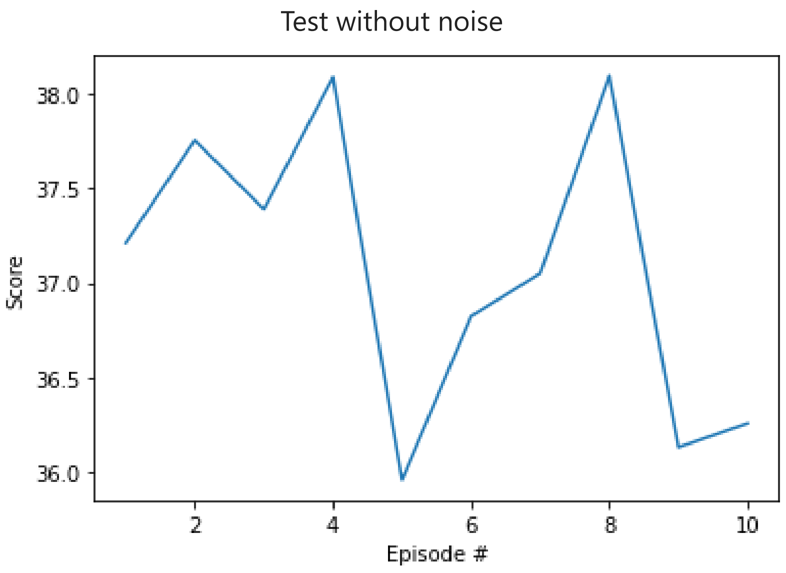
### 1. Two hidden fully connected layers with 64 and 64 units

```
Actor network: Actor(
  (fc1): Linear(in_features=33, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=4, bias=True)
)
Critic network: Critic(
  (fcs1): Linear(in_features=33, out_features=64, bias=True)
  (fc2): Linear(in_features=68, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=1, bias=True)
)
```





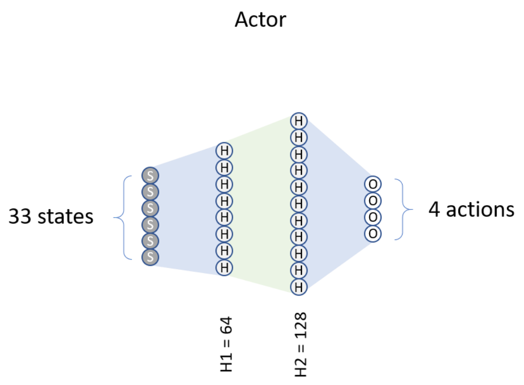


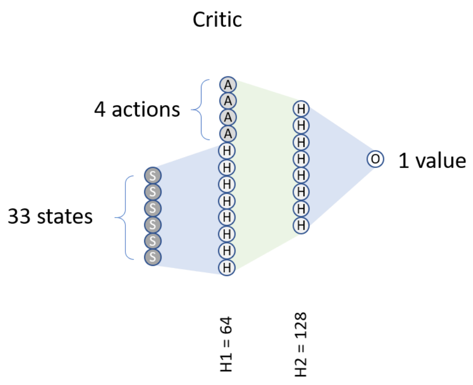


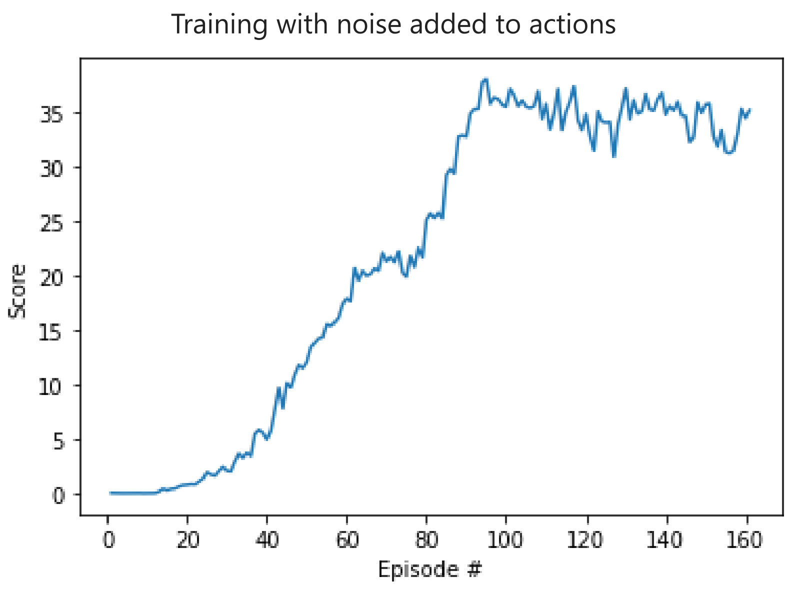
500 episodes needed to solve the environment.

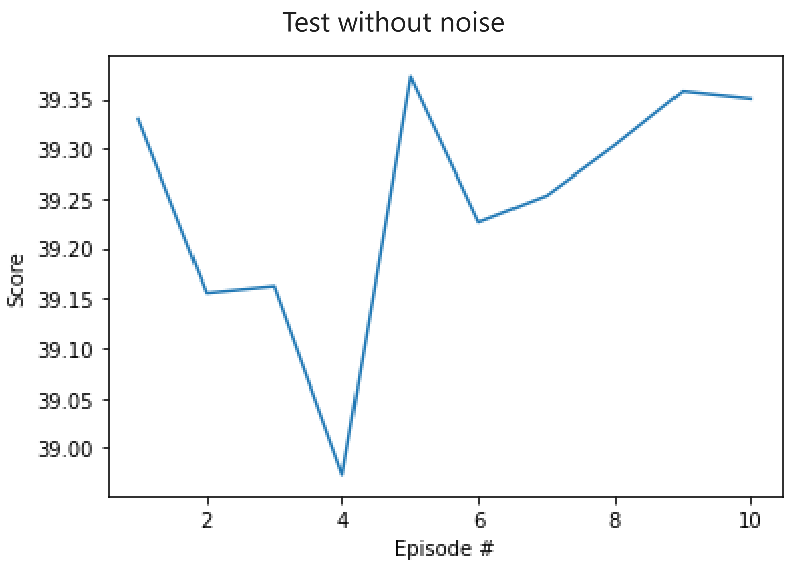
## 2. Two hidden fully connected layers with 64 and 128 units

```
Actor network: Actor(
  (fc1): Linear(in_features=33, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=4, bias=True)
)
Critic network: Critic(
  (fcs1): Linear(in_features=33, out_features=64, bias=True)
  (fc2): Linear(in_features=68, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=1, bias=True)
)
```





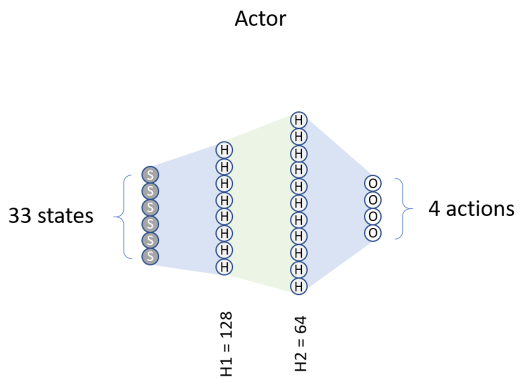


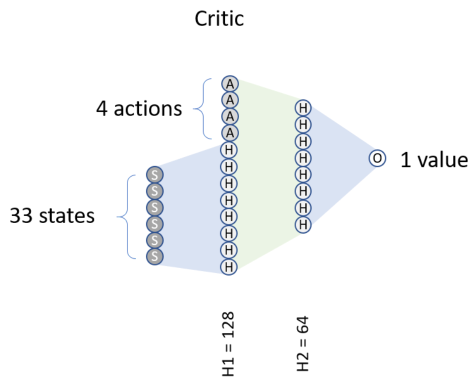


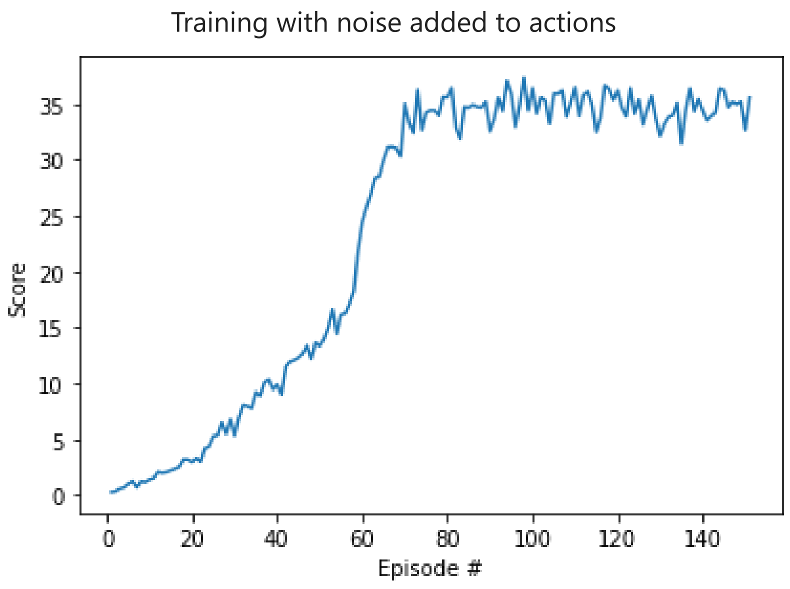
160 episodes needed to solve the environment.

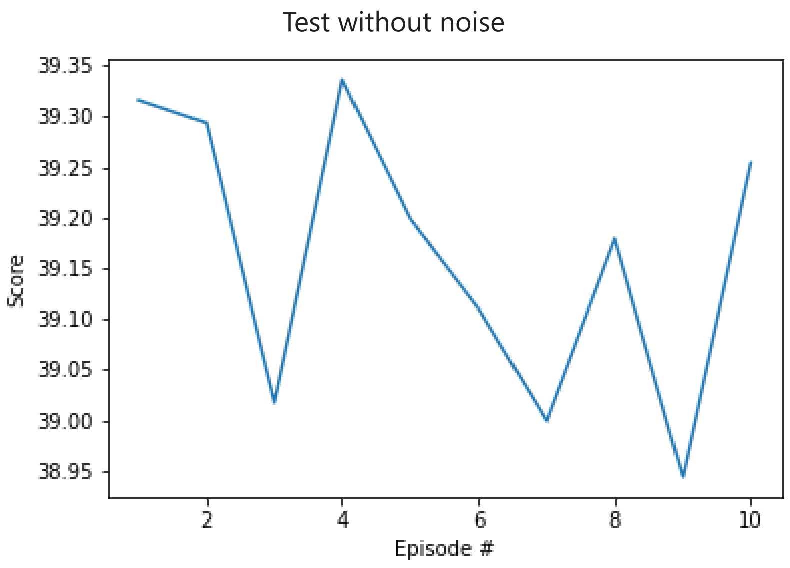
## 3. Two hidden fully connected layers with 128 and 64 units

```
Actor network: Actor(
  (fc1): Linear(in_features=33, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=4, bias=True)
)
Critic network: Critic(
  (fcs1): Linear(in_features=33, out_features=128, bias=True)
  (fc2): Linear(in_features=132, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=1, bias=True)
)
```





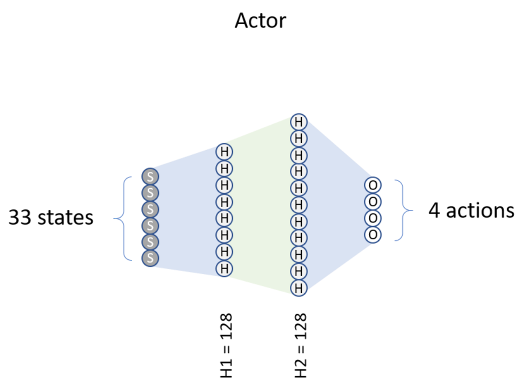


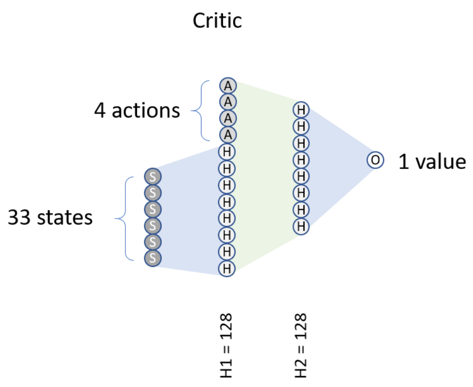


150 episodes needed to solve the environment.

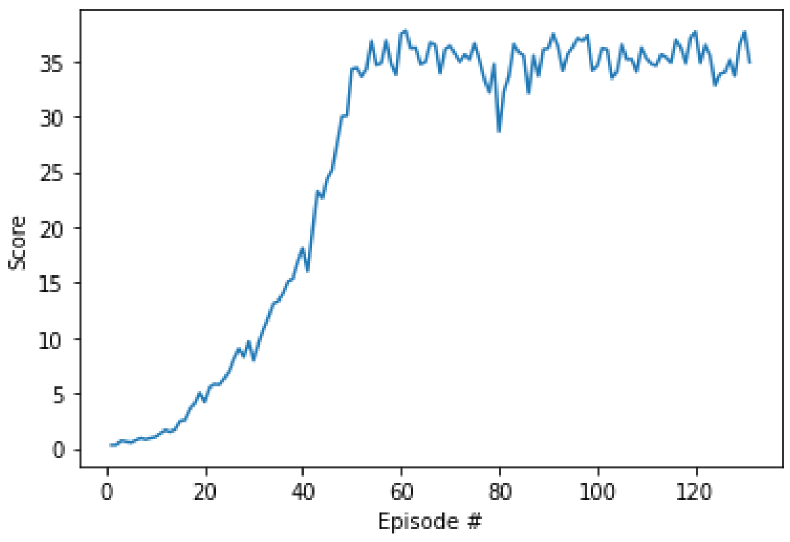
## 4. Two hidden fully connected layers with 128 and 128 units

```
Actor network: Actor(
  (fc1): Linear(in_features=33, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=4, bias=True)
)
Critic network: Critic(
  (fcs1): Linear(in_features=33, out_features=128, bias=True)
  (fc2): Linear(in_features=132, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=1, bias=True)
)
```

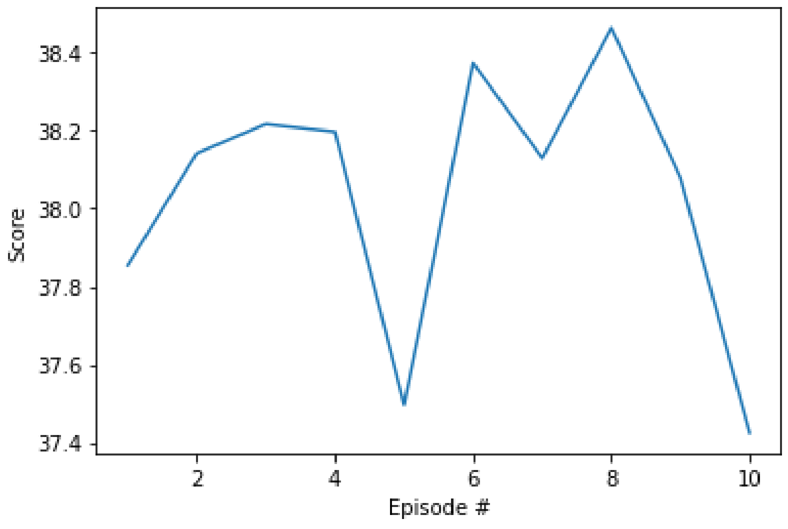




Training with noise added to actions



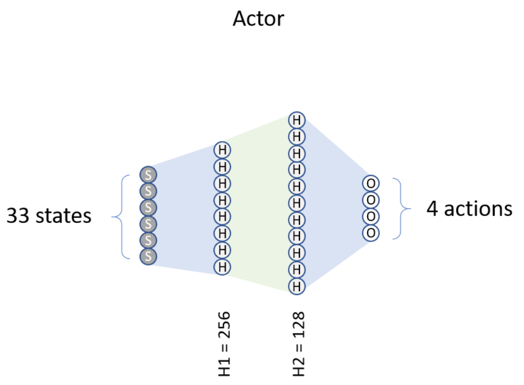
Test without noise

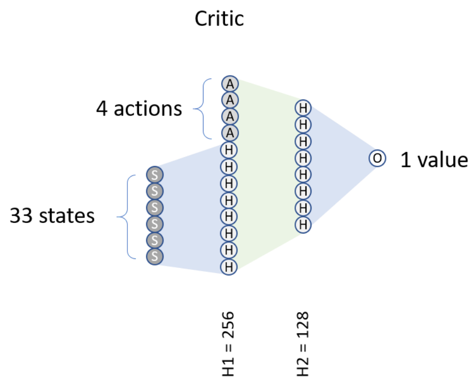


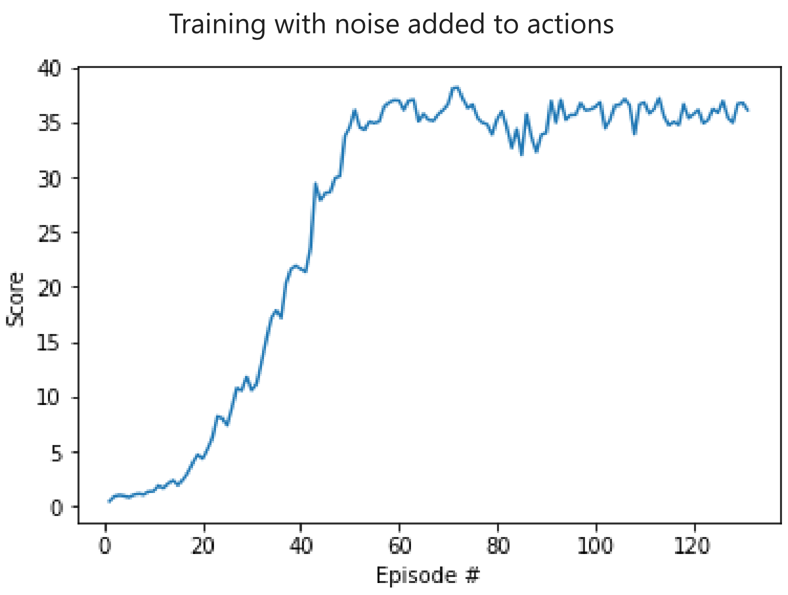
130 episodes needed to solve the environment.

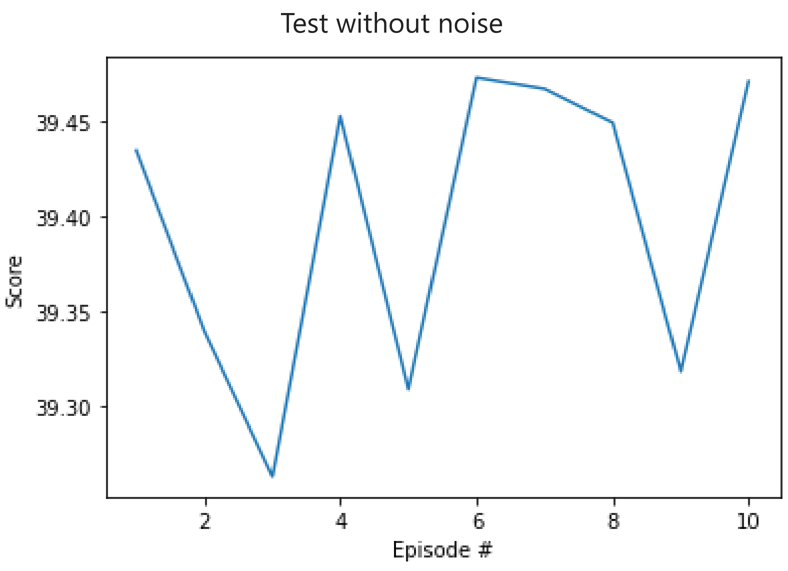
## 4. Two hidden fully connected layers with 256 and 128 units

```
Actor network: Actor(
  (fc1): Linear(in_features=33, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=4, bias=True)
)
Critic network: Critic(
  (fcs1): Linear(in_features=33, out_features=256, bias=True)
  (fc2): Linear(in_features=260, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=1, bias=True)
)
```





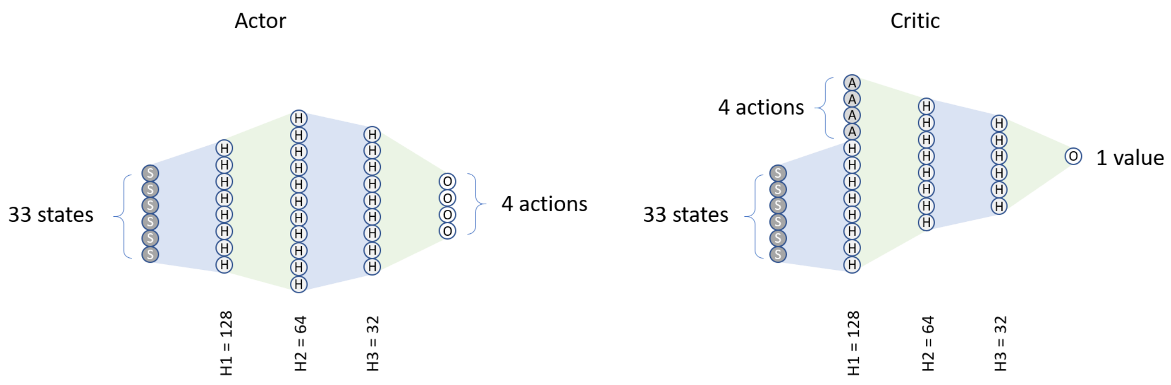




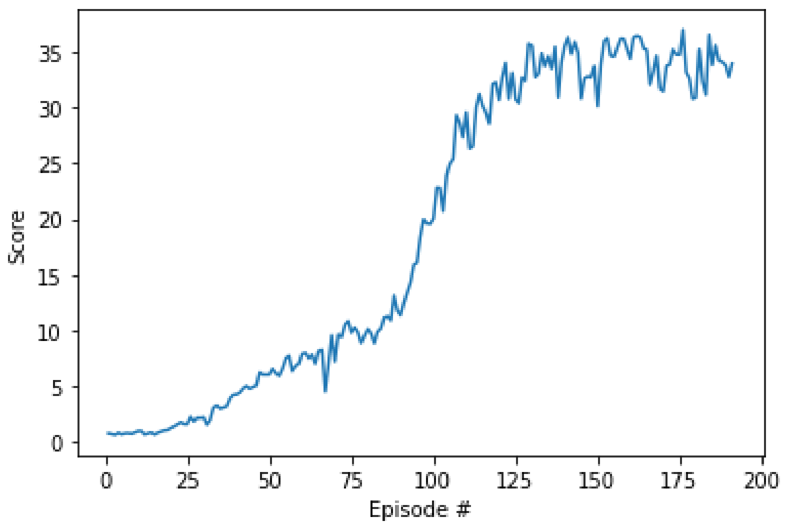
130 episodes needed to solve the environment.

## 5. Three hidden fully connected layers with 128, 64 and 32 units

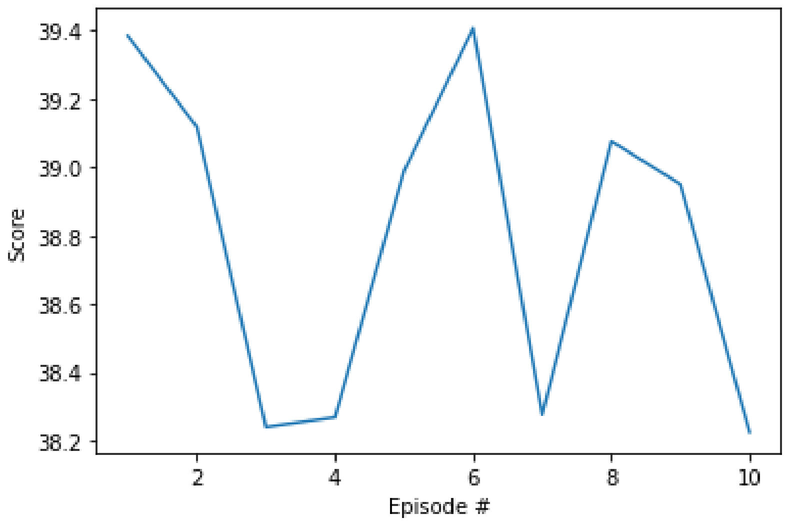
```
Actor network: Actor(
  (fc1): Linear(in_features=33, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=32, bias=True)
  (fc4): Linear(in_features=32, out_features=4, bias=True)
)
Critic network: Critic(
  (fcs1): Linear(in_features=33, out_features=128, bias=True)
  (fc2): Linear(in_features=132, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=32, bias=True)
  (fc4): Linear(in_features=32, out_features=1, bias=True)
)
```



Training with noise added to actions



Test without noise



190 episodes needed to solve the environment.

## Summary, conclusion and discussion

I chose DDPG as the first attempts and it worked quite well. I had some difficulties at the start, but after that the initial problems were resolved and the learning rate was reduced, DDPG worked nicely. I put my focus on the network structure and tried to understand how that influences the learning process. However, the results were not too interesting. It seems like once the network was large enough, it did the work nicely, with number of episodes to reach the target being around 150.

I add one more hidden layer with the hope that nonlinearities would be modeled in a better way. My conclusion is that the case under consideration was simple to be modeled reasonably well with only two layers.

There are authors on internet, who suggest that regularization methods or the decay of the noise are cruicial to solve the environment efficiently. I could not see that. I implmented noise decay, but did not need to use it. Equally, I had no regularization without any problems, which I think makes sense, as the risks for overfitting must be small.

There are other methods than DDPG, which I will try in the future. However, selection of an appropriate network for the problem at hand is an important step as well. A systematic, hopefully automatic study of all hyper parameters at the initial point seems to be a wise starting point. I hope I will use reinforcement learning in a real-world problem in a soon future to go through the process once again.

## File organization

The implementation and test of the method are presented in this repository, with the files organized as follows:

1. `continuous_control_final.ipynb` : Main notebook.
2. `ddpg_agent.py` : Agent definition, memory replay and noise definitions.
3. `model.py` : Model definitions.
4. `checkpoint_actor.pth` and `checkpoint_critic.pth` : Checkpoint data with weights for trained agent.
5. `test_final.ipynb` : Test notebook.
6. `figs` : Folder will figures used in presentations.
7. `videos` : Folder with a recorder result.
8. `Report.html` : Detailed report
9. `three_hidden_layers` : Folder containing model and agent with three hiden layers.